SYSTEMES EMBARQUES

WORKSHOP : POINTEURS

INTRODUCTION

Dans ce workshop, vous allez aborder la notion de pointeur en langage C et découvrir ses trois utilisations majeures :

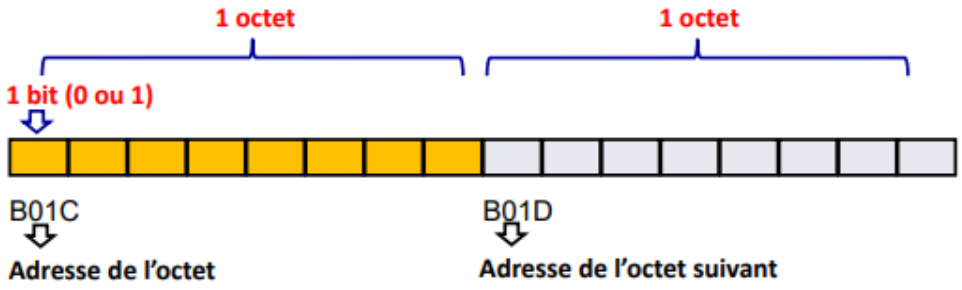
- Le passage des adresses à des fonctions ;
- La manipulation de données complexes
  - Tableaux
  - Chaînes de caractères (cf. Corbeilles d'exercices)
  - Structures de données
- L'allocation dynamique de mémoire.

Table des matières

1. POINTEURS

Rappel : Stockage et adresse

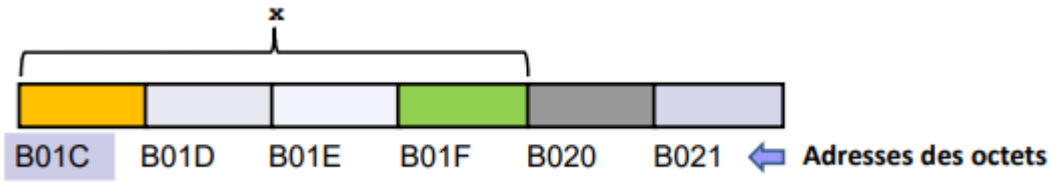
- Dans la mémoire de l'ordinateur, les données sont stockées sous **forme binaire**.
- La mémoire est divisée en « cases » de taille 8 bits, appelées **octets** (bytes en anglais).
- Chaque octet est repéré par son **adresse**, qui est souvent donnée par un nombre hexadécimal.



Introduction aux pointeurs :

```
int x ;
```

- Cette déclaration de variable réalise deux opérations :
  - Définition d'une variable `x` pouvant être utilisée dans le programme pour manipuler des données.
  - Réservation ou allocation d'un espace mémoire où sera stocké le contenu de la variable.
- La variable `x` est stockée sur 4 octets. Son adresse est celle du premier octet, soit dans la figure ci-dessous BO1C.

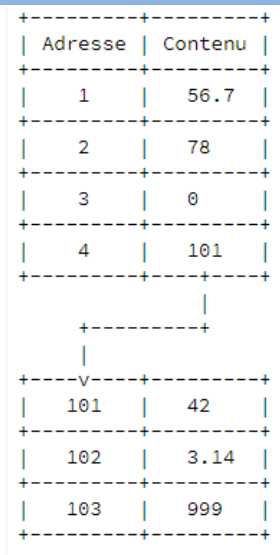


Définition des pointeurs :

```
int x ;
```

- La variable `x` a une adresse en mémoire, dans l'exemple précédent c'est B01C.
- Pour avoir accès à l'adresse de la variable `x`, on utilise l'opérateur **`&`**.
- **`&x`** représente l'adresse de la variable `x`.
- Pour manipuler les adresses, on définit un **nouveau type de variable : les pointeurs.**

> **Un pointeur est une variable qui contient l'adresse d'une autre variable. C'est une adresse typée.**



Comme vous pouvez le voir sur la figure ci-dessus, le contenu à l'adresse 4 est lui-même une adresse et pointe sur le contenu situé à l'adresse 101.

- **Déclaration d'un pointeur** :

```
double *p;
```

  - `p` est un « pointeur sur une variable de type double ».
  - `p` est une adresse typée. C'est l'adresse de la première case mémoire contenant la donnée. On dit que `p` pointe sur une variable.
  - `*p` représente le contenu de la variable pointée par `p`.

- **Initialisation d'un pointeur** :
  - On peut donner l'adresse d'une variable déjà existante :

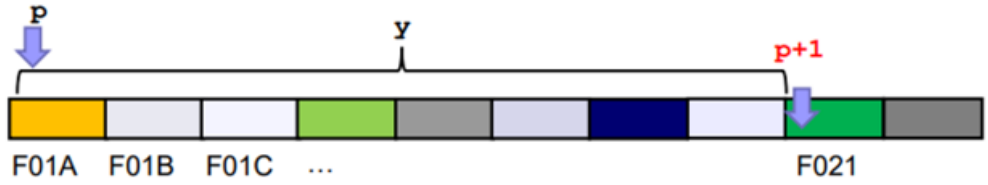
```
double y;
double *p;
p=&y;
```

  - Pour utiliser le pointeur, il faut le faire pointer sur une adresse. Pour cela, on utilise une expression appliquant l'opérateur `&` à une variable.
  - On peut initialiser un pointeur à `NULL`, dans ce cas le pointeur ne contient aucune adresse :

```
int *p = NULL;
```

- **Pointer sur la case suivante** :
  - **`p+1`** pointe sur le double suivant en mémoire.
  - **`*(p+1)`** est le contenu de la variable pointée par **`p+1`**.

```
double y ;
double *p ;
p=&y ;
```



Comme vous pouvez le voir sur la figure ci-dessus :

- la variable `y` stockée sur 8 octets (c'est double)
- L'adresse de `y` est F01A. `p` vaut donc F01A et `p+1` vaut F021.

Exemple

Recopier le code ci-dessous et analyser le résultat retourné :

```
int main(){
    int n;
    int *p;
    n =5;
    printf("%d\n",n);
    p =&n;
    printf("%d\n",*p);
    *p=1;
    printf("%d\n",n);
    printf("%d\n",*p);
    printf("%x %x\n",p,&n);
    printf("%x %x\n",p+1,&n+2 );
return 0;
}
```

*p désigne le contenu de la case mémoire pointée par p

%x est le format pour les adresses en hexadécimal

2. FONCTIONS ET POINTEURS

En langage C, une fonction travaille sur des copies des variables. On parle de « ***passage d'argument par valeur*** ». Un premier usage des pointeurs, c'est pour écrire des fonctions qui modifient des données, pour cela, ces fonctions reçoivent l'adresse de la donnée à modifier, On parle de « ***passage d'argument par adresse*** ».

Exercice : Echanger deux nombres

1. Ecrire une fonction qui échange deux nombres, et qui aura comme paramètres deux entiers a et b :

```
void echanger(int a, int b)
{
      int tmp;          // complétez

}
```

Pour tester la fonction, écrire un `main` qui déclare et affecte deux variables, les échange, et affiche leurs valeurs :

```
int main()
{
      int a=3,b=258;

      echanger(   ,   );  // complétez

      printf("%d\t%d\n",a,b);

      return 0;
}
```

Qu'est-ce que vous remarquez ? Est-ce que `a` **et** `b` **ont été modifiées ?**

2. Essayez maintenant de modifier votre fonction echanger() en passant en paramètres **les adresses de deux entiers** `a` **et** `b` :

```
void echanger(int *adr_a, int *adr_b)
{
   int tmp;          // complétez

}
```

Pour tester, modifiez votre main et analyser le résultat retourné.

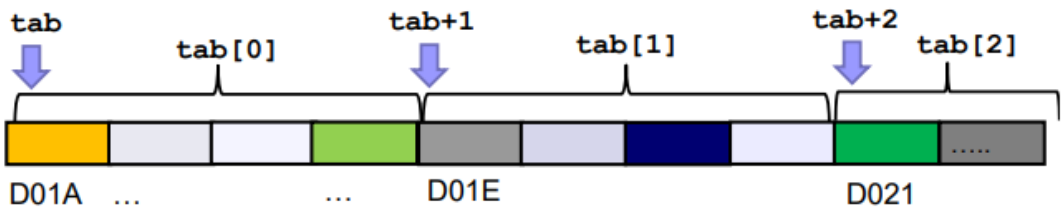
3. MANIPULATION DE DONNEES COMPLEXES

Tableaux et Pointeurs

```
int tab[100];
```

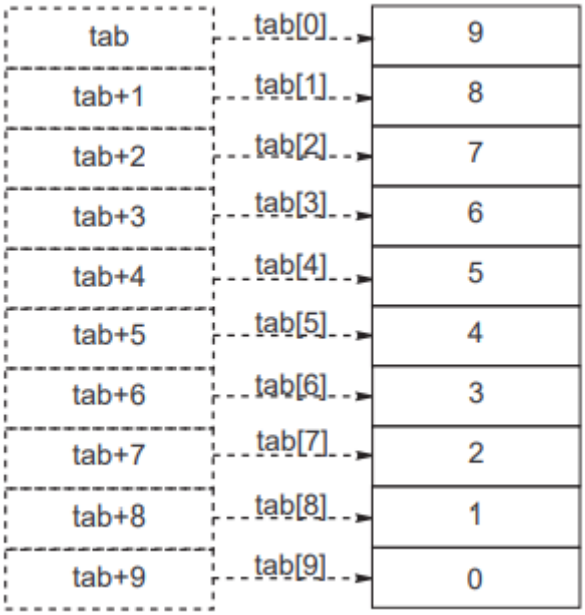
- Cette déclaration fait deux choses :
  - Définition d'une variable pouvant être utilisée dans le programme pour manipuler des données.
  - Allocation d'un espace mémoire de 100 cases contigües de 4 octets.
- En fait, **un tableau n'est rien d'autre qu'un pointeur.**
- La variable `tab` est un **pointeur** qui est l'adresse du premier élément du tableau. En d'autres termes, le nom d'un tableau est l'adresse de son premier élément.

  - `tab est l'adresse de tab[0]`
  - `tab+i est l'adresse de tab[i]`
  - Il y a équivalence entre `tab+i` et `&tab[i]` et entre `*(tab+i)` et `tab[i]`



La figure ci-dessous montre l'espace mémoire correspondant à la définition d'un tableau de dix entiers avec une initialisation selon la ligne :

```
int tab[10] = { 9,8,7,6,5,4,3,2,1,0};
```



Exemple

Recopier le code ci-dessous et analyser le résultat retourné :

```
int main()
{   int tab[10];
    int i;
    for (i=0;i<10;i++) tab[i]=i;
    printf("%d\n",tab[0]);
    printf("%d\n",*tab);
    printf("%d %d\n",*(tab+1),tab[1]);
    printf("%x %x\n",tab,&tab[0]);
    return 0;
}
```

Exercice : Inverser un tableau

Dans cet exercice, vous allez écrire un programme qui permet d'inverser le contenu d'un tableau (le premier élément devient le dernier, l'avant-dernier le deuxième et ainsi de suite). Pour cela :

1. Ecrire une fonction `SaisieTableau()` un tableau `T` d'entier de dimension `N`.
2. Ecrire une fonction `AfficheTableau()` qui affiche le tableau `T`.
3. Ecrire une fonction `InverseTableau()` qui inverse le contenu d'un tableau `T` sans utiliser un tableau d'aide. Pensez à la fonction `void echanger(int *adr_a, int *adr_b)` présentée dans la partie « Fonctions et Pointeurs ».
4. Ecrire le programme principal pour tester vos fonctions.

Structures et Pointeurs

Les structures en langage C, vous permet de créer vos propres types de variables. Une structure est un assemblage de variables qui peuvent avoir différents types. Contrairement aux tableaux qui vous obligent à utiliser le même type dans tout le tableau, vous pouvez créer une structure comportant des variables de types `long`,`char`,`int` et `double` à la fois.

- **Rappel :**

  - **Déclaration d'une structure**

```
struct nomStructure {
    type1 champ1;
    ...
    typeN champN;
} variables;
```

Voici un exemple qui déclare une structure permettant de stocker un nombre complexe :

```
struct complex {
    double reel; /* partie reelle */
    double imag; /* partie imaginaire */
};
```

  - **Accès aux champs** : L'opérateur « . » permet d'accéder directement à l'un des champs d'une structure :

```
struct complex a;

a.reel = 2;
a.imag = 3;
```

  - **Utilisation de `typedef` :** Le mot-clé `typedef` permet d'associer un nom à un type donné.

L'exemple précédent peut donc se réécrire de la manière suivante :

```
typedef struct {
    double reel; /* partie réelle */
    double imag; /* partie imaginaire */
} complexe;

complexe a;

a.reel = 2;
a.imag = 3;
```

- **Pointeurs sur structures :** L'utilisation de pointeurs sur structures est très courante en langage C.

Voici un exemple d'utilisation d'un pointeur sur un complexe :

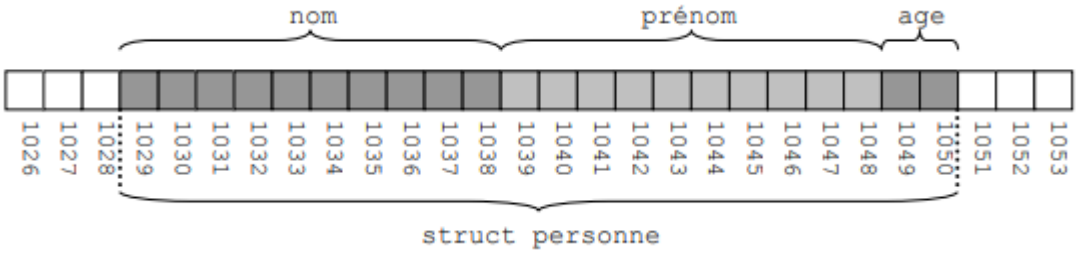
```
complexe a = { 3.5, -5.12 };
complexe *p = &a;
(*p).reel = 1;
(*p).imag = -1;
```

Vous pouvez aussi utiliser l'opérateur « -> » **pour accéder aux champs d'une structure via un pointeur.**
Ecrire `pointeur->champ` est donc STRICTEMENT équivalent à écrire `(*pointeur).champ`

- **Les structures en mémoire**

En mémoire, les structures présentent également des similitudes avec les tableaux. Les champs qui composent une structure sont en effet placés de façon *contigüe* en mémoire, exactement comme les éléments formant un tableau.
Exemple : Voici dans la figure ci-dessous la représentation de la structure personne suivante :

```
struct personne{
      char nom[10];
      char prenom[10];
      short age;
};
```



La différence avec les tableaux est qu'ici, les éléments sont hétérogènes, dans le sens où ils peuvent avoir des types différents. Donc, l'espace occupé par chaque élément n'est pas forcément le même. Dans l'exemple ci-dessus, le nom et le prénom occupent ainsi chacun `10 octets` car il s'agit de tableaux de 10 caractères, mais l'âge n'en occupe que 2, car c'est un entier `short`.

Exercice : Structure Etudiant

Soit la structure « `Etudiant`» suivante où un étudiant est représenté par son nom, ses notes et son âge.

```
typedef struct
{
    char nom[20];
    float moy[3];
    int age;
} T_etud;
```

1. Ecrire une fonction `saisir_donnees()` permettant de saisir les données d'un étudiant. **Le paramètre doit être passé par adresse.**

2. Ecrire une fonction `calculer_moy_generale()` qui retourne un `float`. **Le paramètre doit être passé par adresse.**

3. Ecrire une fonction `afficher()` pour afficher à l'écran un étudiant, par exemple :

```
Nom : Dupont
Age : 18 ans
Moyennes de l'étudiant : 12.50 8.00 10.25
```

4. Ecrire le programme principal pour tester votre programme.

4. ALLOCATION DYNAMIQUE ET POINTEURS

Dans cette partie nous allons aborder la troisième et dernière utilisation majeure des pointeurs : l'allocation dynamique de mémoire.

L'allocation dynamique de mémoire permet en cours d'exécution d'un programme de réserver un espace mémoire dont la taille n'est pas nécessairement connue lors de la compilation.

Les principales fonctions d'allocation dynamiques sont :

- `Malloc()` pour allouer un bloc de mémoire
- `Calloc()` pour allouer un bloc de mémoire et l'initialiser à 0
- `Realloc()` pour agrandir la taille d'un bloc de mémoire
- `Free()` pour libérer un bloc de mémoire

Ces fonctions se trouvent dans la bibliothèque standard `<stdlib.h>`.

Les prototypes de ces quatre fonctions sont les suivant :

- « `void* malloc (size_t size)` » : alloue size octets, et renvoie un pointeur sur la mémoire allouée. Le contenu de la zone de mémoire n'est pas initialisé.
- « `void* calloc (size_t nmemb, size_t size)` » : alloue la mémoire nécessaire pour un tableau de `nmemb` éléments, chacun d'eux représentant `size` octets, et renvoie un pointeur vers la mémoire allouée. Cette zone est remplie avec des zéros.
- « `void* realloc (void* ptr, size_t size)` » : modifie la taille du bloc de mémoire pointée par `ptr` pour l'amener à une taille de `size` octets. `realloc()` réalloue une nouvelle zone mémoire et recopie l'ancienne dans la nouvelle sans initialiser le reste.
- « `void free (void* ptr)` » : libère l'espace mémoire pointé par `ptr`, qui a été obtenu lors d'un appel antérieur à `malloc()`, `calloc()` ou `realloc()`.

Exemple :

Le programme ci-dessous alloue un tableau de dix `int` et utilise `realloc()` pour agrandir celui-ci à vingt `int`. Recopier le programme et analyser le résultat retourné :

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int i;
    int *p = malloc(sizeof(int[10]));

    if (p == NULL)
    {
        printf("Echec de l'allocation\n");
        return -1;
    }

    for (i = 0; i < 10; ++i)
```

```
        p[i] = i * 10;

    int *tmp = realloc(p, sizeof(int[20]));

    if (tmp == NULL)
    {
    free(p);
        printf("Echec de l'allocation\n");
        return -1;
    }

    p = tmp;

    for (i = 10; i < 20; ++i)
        p[i] = i * 10;

    for (i = 0; i < 20; ++i)
        printf("p[%d] = %d\n", i, p[i]);

    free(p);
    return 0;
}
```

Exercice : Tableaux de caractères dynamiques

Ecrire un programme qui lit 10 phrases d'une longueur maximale de 200 caractères au clavier et qui les mémorise dans un tableau de pointeurs sur `char` en réservant **dynamiquement** l'emplacement en mémoire pour les chaînes. Ensuite, l'ordre des phrases est inversé en modifiant les pointeurs et le tableau résultant est affiché.